REMARKS/ARGUMENTS

In paragraph 2 of the Office action, the examiner states that the Information Disclosure Statement filed 10/20/2003 did not contain the required legible copy of each non-patent literature publication. In response, copies of the two publications listed on the Information Disclosure Statement filed 10/20/2003 are filed herewith.

In paragraph 4 of the Office action, claims 1-7, 11-14, 17, and 20 stand provisionally rejected on the ground of nonstatutory, obviousness-type, double patenting as being unpatentable over claims 1-7, 12-15, 18, and 21 of copending Application No. 10/689,355. In paragraph 7 of the Office action, claims 1-9, 11-17, and 20 stand provisionally rejected on the ground of nonstatutory, obviousness-type, double patenting as being unpatentable over claims 1-8, 12-15, and 20 of copending Application No. 10/689,336. Because both of these rejections are provisional, these rejections will be dealt with when patentable subject matter is indicated.

In paragraphs 10 and 11 of the Office action, claim 20 stands rejected under 35 U.S.C. § 101 for reciting "a memory device." In response, claim 20 has been amended to recite "a computer readable memory device." Claims to a "computer readable medium" are authorized in the Interim Guidelines for Subject Matter Eligibility, in the section dealing with "practical application." It is believed that claim 20, as amended, is in compliance with the interim guidelines such that the 35 U.S.C. § 101 rejection should be withdrawn.

In paragraph 13 of the Office action, claims 1-20 stand rejected under 35 U.S.C. § 112, second paragraph, as being indefinite for failing to particularly point out and distinctly claim the subject matter which applicant regards as the invention. In paragraph ai, the examiner states that claims 1, 11, and 20 recite "determining a running partial deviation sum for each of said plurality of processing elements." The examiner states that it is unclear how this step relates to the rest of the invention. Each of claims 1, 11, and 20 has been amended to state how the running partial deviation is used in the next step of the claim. In addition, other changes have been made to claims 1, 11, and 20 for reasons of readability and to more closely tie the elements of the claims together, and not for reasons related to patentability.

In paragraph aii, claims 4 and 13 recite "*V*". It is the examiner's position that it is unclear what is meant by the "*V*". Each of claims 4 and 13 has been amended to recite that "*V*" is the total number of tasks. The examiner also indicates, with respect to "E_r", that it is unclear how that value is determined for each of the plurality of processing elements. The examiner's attention is respectfully directed to paragraph [0045] of the application as filed which provides:

> In the current embodiment, each PE is assigned a different E_r value for controlling the rounding. The simplest form for the function E is the case in which $E_r = P_r$, where P_r represents the PEs position in the loop. For example, for PE_0, $E_0 = 0$; for PE_1, $E_1 = 1$; for PE_2, $E_2 = 2$; etc. By assigning each PE in the loop a different E_r value, the rounding function can be controlled such that some of the local means are rounded up and some of the local means are rounded down, thus insuring that $V = \sum_{i=0}^{i=N-1} M_i$. It should be noted that in the current embodiment, the local mean for each PE 30 in the loop is computed in parallel with the local means of the other PEs in the loop.

It is submitted that reading claims 4 and 13 in view of the disclosure of paragraph [0045], one of ordinary skill in the art would understand how the value E_r is derived for each of the plurality of processing elements. Finally, with respect to claim 13, a definition has been provided for PE_r.

In paragraph aiii, the examiner indicates that it is unclear in claim 5 how E_r "controls" the *Trunc* function. The language of claim 5 and claim 14 has been amended to recite that the *Trunc* function is responsive to the value of E_r. With respect to the examiner's question about how this step is possible, "since each E_r value is set ahead of time and must be different for each processing element," the examiner's attention is respectfully directed to paragraph [0045] reproduced above.

With respect to paragraph aiv, the examiner states that the recitation of "X and (X+1)" is unclear. The examiner's attention is respectfully directed to paragraph [0014] of the specification which provides as follows:

> The present invention enables tasks to be distributed along a group of serially connected PEs so that each PE typically has X number of tasks or (X+1) number of tasks to perform in the next phase. The present invention may be performed using the hardware and software (i.e., the local processing capability) of each PE within the array. Those advantages and benefits, and others, will become apparent from description of the invention below.

The examiner's attention is also directed to the table appearing in paragraph [0046] which provides:

PE_r	v_r	E_r	$(V+E_r)/N$	$M_r = Trunc((V+E_r)/N)$	D_r
PE_0	3	0	5.375	5	-2
PE_1	6	1	5.5	5	1
PE_2	2	2	5.625	5	-3
PE_3	7	3	5.75	5	2
PE_4	8	4	5.875	5	3
PE_5	5	5	6.0	6	-1
PE_6	5	6	6.125	6	-1
PE_7	7	7	6.25	6	1

Table #1 – Local Mean Calculation for the Loop 50 ($V = 43$, $N = 8$).

The language of claim 6 has been amended to indicate that a local mean for each group is equal to either X, or X+1, as seen clearly from Table No. 1 where X = 5 and X+1 = 6.

In paragraph av, the examiner asserts that certain of the language of claim 8 is unclear. Claims 8 and claim 15 have been amended to clearly identify that the local deviation associated with each of the plurality of processing elements is sent to an adjacent processing element.

In view of the foregoing, it is respectfully requested that the rejection of claims 1-20 under 35 U.S.C. § 112, second paragraph, be withdrawn.

In paragraph 15 of the Office action, claims 1-20 stand rejected under 35 U.S.C. § 103(a) as being unpatentable over Smith (U.S. Pub. No. 2004/0024874) in view of Wheat (U.S. Patent No. 5,630,129). Applicant respectfully traverses that rejection.

It is the examiner's position that both Smith and Wheat teach methods of load balancing. Although that assertion is correct, the methods of load balancing taught by Smith and Wheat are so different from one another, and different from what is claimed, that it is not possible for the combination of those two references to suggest the claimed invention.

The examiner asserts that Smith teaches "calculating a local deviation for each of said plurality of processing elements." Claims 1 and 20 have been amended to recite that the local deviation is calculated "from said local mean number." Claim 11, as originally presented, contained similar subject matter. Smith does not teach calculating a local deviation from said local mean number because, as the examiner recognizes, Smith does not teach calculating a local mean number. Accordingly, the "calculating a local deviation from said local mean number" is not met by Smith.

The examiner next asserts that Smith discloses "determining a clockwise transfer parameter and an anti-clockwise transfer parameter for each of said plurality of processing elements" citing paragraphs 18-20 of Smith. Smith does not calculate a transfer parameter as asserted by the examiner. Smith merely compares the workloads of pairs of processors, and the processor having the lower level of work simply requests work from the processor having a higher level of work. There is no transfer parameter calculated. This is made clear in Smith, paragraph [0038] which recites in part:

> With a uni-directional link from processor A 13 ("upstream") to processor B 14 ("downstream"), A informs B of how much workload it has, B then compares this with its own level of workload, and if B is less loaded than A, then it requests work from A. It is therefore ensured that B has at least as much work as A. Such pairs are linked end to end in a chain, with all the links going in the same direction, with the ends of the chain joined together. This forms a closed loop with all the workload transfers travelling in the same direction. Since in each pair the one downstream of the link has at least as much work as the one upstream, and every processor in every pair downstream of another processor, it ensures that the entire ring is inherently balanced.

Finally, the examiner asserts that Smith teaches "redistributing tasks among said plurality of processing elements in response to said clockwise transfer parameter and said anti-clockwise parameter for each of said plurality of processing elements" citing paragraph [0020] of Smith. As discussed above, Smith does not operate in such a manner so as to generate transfer parameters. Accordingly, the redistribution does not take place in response to transfer parameters. Redistribution takes place in Smith in response to a request from one processing element, having a lighter workload, made to another processing element having a heavier workload.

It is thus seen that Smith, although it does deal with load balancing, operates in such a completely different manner that it discloses none of the steps of claims 1, 11, and 20.

The defects of the primary reference to Smith cannot be overcome by Wheat. The examiner asserts in paragraph 19 that "Wheat teaches a dynamic load balancing method by determining the average load across a processor array and minimizing a global imbalance or workloads within a finite number of balancing steps" citing column 6, lines 58-67. Although the cited portion of Smith does discuss a "global imbalance," this is part of a discussion in which Wheat proves that his method minimizes the global imbalance. The actual method is set forth beginning in column 5, line 50, with a determination of workloads. Workloads are then compared amongst processors. See column 5, lines 60-67, which provide:

> Each processor compares its work load to the work load of the other processors in its neighborhood and determines which processors have greater work loads than its own. If any are found, it selects the one with the greatest work load (ties are broken arbitrarily) and sends a request for work to that processor. Each processor may send only one work request, but a single processor may receive several work requests.

Transfers take place according to priorities as discussed in column 6, lines 40-57, which provide as follows:

> FIG. 4 illustrates an example of element priorities and selection for exporting four elements to the east neighboring processor. Initially, elements 3, 6, 9, and 12 are eligible for export. Their priorities are computed; element 3, for example, has priority -2, since it has two local neighbors (-2), one neighbor in a concerned partner processor (-2), and one neighbor in the importing processor (+2). Elements 6 and 9 share the highest priority, but since element 6 has a greater work load, it is selected. Element 5 becomes eligible for export, but its priority is low since it has three local neighbors. The priorities are adjusted, and element 9 is selected, making element 8 a candidate. The priorities are again updated, and the selection process continues with elements 3 and then 12 being selected. Although the work request is not completely satisfied, no other elements are exported, as the work loads of the elements with the highest priority, 5 and 8, are greater than the remaining work request

It is seen that Wheat, although disclosing a method for dynamic load balancing, teaches a very different method than either the claimed invention or Smith. Processor work requests are determined based on processors comparing their workloads with other processors. Requests are then made and granted on the basis of priorities. There is no calculating a local mean number of tasks, determining a running partial deviation, determining transfer parameters, or the other steps of claims 1, 11, and 20. It is respectfully submitted that the load balancing techniques of Smith and Wheat are so dissimilar from one another, and so dissimilar from the claimed invention, that no possible combination of the teachings of the two references renders obvious claims 1-20. For the foregoing reasons, applicant respectfully requests that the 35 U.S.C. § 103 rejection based on the combination of Smith and Wheat be withdrawn.

With respect to paragraphs 24 and 25 of the Office action and claim 4, the Office relies upon official notice for the proposition that "it is well known to perform a local mean calculation using this method and using a truncation function to remove unnecessary decimals." While such a function may be well known in the art, the methods of Wheat and Smith are complete in themselves The effort to graft an unnecessary step onto the methods of Wheat and Smith through the use of official notice is nothing more than an improper hindsight reconstruction. The rejection of claim 4 should be withdrawn.

With respect to paragraphs 26 and 27 and claim 5, the Office provides no basis for the conclusion that "it would have been obvious . . . to change the value of E_r." The examiner provides no basis for where that teaching is found, no citation to the art of record and no reliance upon official notice. The Office appears to be using applicant's disclosure as a basis for a hindsight reconstruction of claim 5.

With respect to paragraphs 28 and 29 of the Office action and claim 6, neither Smith nor Wheat teaches calculating a local mean. Thus, it is difficult to understand how the references can disclose the details of claim 6 when the broad concept of calculating a local mean is not even disclosed.

With respect to paragraphs 30 and 31 of the Office action and claim 7, because neither Smith nor Wheat discloses determining a local mean, one of ordinary skill in the art would not be interested in calculating a local deviation, by any means. The Office appears to be using applicant's disclosure as a basis for a hindsight reconstruction of the claim.

With respect to paragraphs 32-35 of the Office action and claims 8 and 9, the examiner provides no basis for where the missing teachings are found; there is no citation to the art of record and no reliance upon official notice. The Office appears to be using applicant's disclosure as a basis for a hindsight reconstruction of claims 8 and 9.

With respect to paragraphs 36 and 37 of the Office action and claim 10, neither Smith nor Wheat disclose determining clockwise and anti-clockwise transfer parameters. Given that recognition, it is difficult to understand how the references can disclose the details of claim 10 when the broad concept of transfer parameters is not even disclosed.

The same arguments presented above are applicable to claims 11-19.

Request for Interview

Applicant has made a diligent effort to place the instant application in condition for allowance. If the examiner is of the opinion that the instant application is in condition for disposition other than through allowance, the examiner is respectfully requested to contact applicant's attorney at the telephone number listed below **so that an interview may be scheduled before the issuance a final Office action rejecting the claims**.

Respectfully submitted,

Edward L. Pencoske
Reg. No. 29,688
Jones Day
One Mellon Center
500 Grant Street, Suite 3100
Pittsburgh, PA, USA, 15219
(412) 394-9531
(412) 394-7959 (Fax)
Attorneys for Applicant

Leveraging Cache Coherence in Active Memory Systems

Daehyun Kim, Mainak Chaudhuri, and Mark Heinrich
Computer Systems Laboratory
Cornell University
Ithaca, NY 14853
{daehyun,mainak,heinrich}@csl.cornell.edu

ABSTRACT

Active memory systems help processors overcome the memory wall when applications exhibit poor cache behavior. They consist of either active memory elements that perform data parallel computations in the memory system itself, or an active memory controller that supports address re-mapping techniques that improve data locality. Both active memory approaches create coherence problems—even on uniprocessor systems—since there are either additional processors operating on the data directly, or the processor is allowed to refer to the same data via more than one address. While most active memory implementations require cache flushes, we propose a new technique to solve the coherence problem by extending the coherence protocol. Our active memory controller leverages and extends the coherence mechanism, so that re-mapping techniques work transparently on both uniprocessor and multiprocessor systems.

We present a microarchitecture for an active memory controller with a programmable core and specialized hardware that accelerates cache line assembly and disassembly. We present detailed simulation results that show uniprocessor speedup from 1.3 to 7.6 on a range of applications and microbenchmarks. In addition to uniprocessor speedup, we show single-node multiprocessor speedup for parallel active memory applications and discuss how the same controller architecture supports coherent multi-node systems called active memory clusters.

Categories and Subject Descriptors

B.3.m [**Memory Structure**]: Miscellaneous;
C.1.4 [**Processor Architectures**]: Parallel Architectures

General Terms

Performance, Design

Keywords

Active Memory, Cache Coherence, Address Re-mapping

ICS'02, June 22-26,2002, New York, New York, USA.

1. INTRODUCTION

One of the most significant challenges facing computer architects today is overcoming the memory wall [26]. While techniques like prefetching or improvements in the cache hierarchy can reduce memory stall time, there remain classes of applications that are not amenable to these methods and have poor cache behavior. A promising approach to overcoming the memory wall in these applications is the use of active memory systems, where data-parallel computations or scatter/gather operations invoked via address re-mapping techniques are performed in the memory system to either offload computation directly or to reduce the number of processor cache misses.

Both active memory approaches create coherence problems—even on uniprocessor systems—since there are either additional processors in the memory system operating on the data directly, or the main processor is allowed to refer to the same data via more than one address. As we will discuss in Section 1.1, most active memory system approaches require the programmer to insert cache flushes before invoking active memory operations to avoid correctness problems. Cache flush overhead on modern processors can be large (typically requiring a trap to the operating system to execute a privileged instruction or set of instructions) and grows more costly as the number of cache levels increases. Though user-level cache flushes may reduce this overhead, either compilers must conservatively insert flushes to maintain correctness, or inserting flushes requires human intervention. Further, this software cache-coherent programming model via flushes faces even larger difficulties on popular single-node multiprocessor servers (SMPs). With process migration in a general-purpose SMP environment, even a uniprocessor active memory application must flush *all* the caches in the system to guarantee correctness, not just its own. In addition, we will describe active memory techniques that require coherence for correctness and for which flushes of any kind are insufficient.

We propose an active memory system that leverages and extends the hardware cache coherence protocol already present on both uniprocessor (for coherent I/O) and multiprocessor systems to provide improved performance on a range of applications. Our focus in this work is on an active memory controller that supports address re-mapping techniques to improve processor cache behavior, though the approach does not preclude the future use of active memory elements as well. The key to the approach is that the active memory controller not only performs the re-mapping operations required, but also runs the directory-based coherence protocol and hence controls which mappings are present in the processor caches. While many machines employ snoopy-based coherence mechanisms, recent architectures [5, 18] have abandoned bus-based snooping in favor of directories because of the decrease in local memory access time and the electrical advantages of point-to-point links between the processor and the memory controller. Because we modify only the memory controller, our technique works with commodity microprocessor and memory technologies. Further, since we leverage the cache coherence mechanism, our active memory techniques work transparently on either uniprocessor or multiprocessor systems.

It is the programmability of our active memory controller combined with specialized hardware that accelerates cache line assembly and disassembly that allows us to extend the cache coherence protocol to improve "traditional" active memory applications that perform matrix transposes and sparse matrix operations. But this same flexibility allows us to improve other classes of applications not usually addressed by active memory systems. In this paper, we also show how we can improve applications that perform repeated linked-list traversals with techniques similar to those in memory forwarding [19] (complete with the "safety net"), but without the processor modifications suggested there. Through detailed simulation of our active memory system we show uniprocessor speedup from 1.3 to 7.6 across these applications. In addition to uniprocessor speedup, we show how our active memory controller improves the performance of parallel applications on single-node multiprocessors, including FFT and parallel reduction.

The rest of the paper is organized as follows. We compare our approach to other active memory approaches in Section 1.1. We describe examples of applications that can benefit from active memory systems and our coherence-based approach in Section 2. In Section 3, we detail the microarchitecture of our active memory controller and describe the functionality of the architecture through illustrative examples. In Section 4 we discuss the applications and benchmarks in our performance study, as well as our simulation methodology. In Section 5 we present simulation results of both uniprocessor and multiprocessor applications on our active memory system compared to the same applications on normal (non-active) memory systems. We also compare the performance of our approach to that of using explicit cache flushes, where it is possible to use flushes at all. In addition, we examine the effect of technology scaling on our approach as well as the performance of some of the microarchitectural features of our active memory controller in isolation. In Section 6 we discuss future work, and Section 7 summarizes our approach and concludes the paper.

1.1 Related Work

Previous work in active memory systems can be divided into projects with active memory elements and those with active memory controllers. The DIVA [8], Active Pages [25], and FlexRAM [13] projects all involve active memory elements—adding processing capability to memory chips, creating so-called PIMs. The application focus of each of these projects is on finding data parallel or streaming operations that can be performed in the memory system, offloading computation from the main processor. The FlexRAM project has also shown speedup for SPEC applications [31, 34]. While our active memory systems approach supports active memory elements, the focus of this paper is solely on our active memory controller design. Both DIVA and FlexRAM have programming models that require cache flushes when communicating between the main processor and the active memory elements. Active Pages initially required cache flushes as well, but realized the critical role of coherence in active memory systems [14] at the same time we did [20], noting that coherence was a better mechanism than flushing for Active Pages. However, the Active Pages project examined coherence only as a mechanism for ensuring the active pages acted on the latest copy of the data, and not in support of the address re-mapping techniques discussed here.

More closely related to this work is the Impulse memory controller [2], which is a hard-wired memory controller that supports a fixed set of address re-mapping techniques to improve processor cache behavior. The Impulse controller improves uniprocessor performance on some of the same applications we use in this paper, namely matrix transpose and scatter/gather operations [36]. However, unlike our active memory approach that leverages cache coherence, the Impulse programming model requires cache flushes when transitioning between normal-space and active-space accesses. The necessity of using cache flushes also complicates the use of these techniques on multiprocessors even for uniprocessor applications (as described earlier).

The main difference between our active memory approach and others is that we leverage, integrate with, and extend the existing hardware cache coherence protocol. With this approach our active memory controller transparently supports address re-mapping techniques on uniprocessor as well as multiprocessor systems. To our knowledge, this paper is the first to present multiprocessor results for applications using active memory re-mapping techniques. In addition, one of the goals of our approach is to use the flexibility of our active memory controller to support new classes of active memory operations like linked-list linearization, parallel reduction, and future applications as the area of active memory systems matures.

2. ACTIVE MEMORY OPERATIONS

In this section we discuss the four classes of active memory operations used in this paper: *Matrix Transpose*, *Sparse Matrix*, *Linked List Linearization* and *Memory-side Merge*. We show why a cache coherence problem arises with these operations, and explain how we solve the problem in our active memory system.

2.1 Matrix Transpose

Consider a matrix A stored in memory in row-major order. If the processor wants to access the matrix A in a column-major fashion, it results in poor cache behavior if the matrix does not fit in the cache. Our active memory controller provides an *in-memory transpose* to solve this problem. An address re-mapping technique is used to map A^T to an additional physical address space A', called the *shadow space*. The shadow space is not backed by any real physical memory, instead it acts as a trigger for the active memory controller. On a shadow space reference the active memory controller gathers individual elements from the normal space, packs them together into a single cache line, and returns it to the processor. Therefore, accesses to A in column-major order can be converted to row-major accesses to the shadow space A', resulting in good cache behavior. In addition, this transformation makes it easy to prefetch the shadow space accesses, which now exhibit good spatial locality.

This matrix transpose operation gives rise to a coherence problem between the original matrix A and the shadow space A'. Any two corresponding elements of the matrix A and the shadow space A' should be consistent with each other, yet the processor may be caching them at two separate locations. One way to ensure coherence is to guarantee that only one of the two spaces is cached at any time. We extend the coherence protocol and treat the access to the two spaces by the same processor in precisely the same way a coherence protocol treats accesses to the same cache line by different processors in a multiprocessor. When returning shadow space cache lines we invalidate the corresponding normal space cache lines from the processor cache. When the processor next references these lines in the normal space, we have guaranteed that this access will cause a cache miss, and our active memory controller can undo the previous remapping operation, returning the latest copy of the data to the processor and invalidating the corresponding shadow space cache lines.

2.2 Sparse Matrix

In this technique the central idea is to gather scattered data that the main processor wishes to access closely spaced in time and assemble them into cache lines. As an example we show the basic loop of Sparse Matrix Vector Multiply (SMVM), using the Compressed Row Storage (CRS) representation of a sparse matrix.

```
for i=0 to N-1
  for j=Arow[i] to Arow[i+1]-1
    v[i] += A[j]*v[Acol[j]];
```

The scattered accesses to the dense vector v will experience cache misses if v is large. To improve cache behavior we re-map v to a shadow space vector $\_v$ and in the loop replace $v[Acol[j]]$ by $\_v[j]$. Whenever the active memory controller sees accesses to $\_v$ it calculates the index j, accesses the cache line containing $Acol[j]$, assembles the corresponding elements of $v[Acol[j]]$ into a cache line and returns that cache line to the main processor. As a result, the main processor sees contiguous accesses to $\_v$ and an improved cache hit rate. This technique again makes it possible to prefetch shadow space accesses to $\_v$.

Here the coherence problem arises between v and $\_v$. The solution is similar to that for matrix transpose, though this particular technique prohibits writes to the shadow space vector $\_v$ because a single cache line of $\_v$ may contain the same element of v more than once. Therefore if the processor writes to one element, the other element (at a different position in the same cache line) will have a stale value. This restriction applies to any active memory implementation of this operation, whether using cache flushes or leveraging the coherence protocol. However, none of the sparse matrix applications that we have seen need to write to the shadow space.

2.3 Linked List Linearization

Searching, inserting, or deleting items in a linked list may require walking through the list and these linked list traversals can exhibit poor cache behavior. The central idea of this active memory technique is to pack consecutive nodes of a linked list into a contiguously-allocated memory region in a dynamic fashion. One *linearize* call to the active memory controller packs a certain number of nodes in the list into a contiguous region, updating the "next" pointers in the list as it goes. The next time the processor traverses the list it sees contiguous memory accesses and hence improved cache behavior. Note that after linearizing the list it is possible to prefetch consecutive nodes of the list, which is difficult in the random linked list structure of the original list.

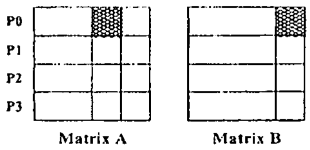

Figure 1: Dense Matrix Multiply

Linearizing linked lists can be done in software without the use of active memory systems. However, a correctness problem arises if after linearization the processor dereferences a dangling pointer that points into the "old" linked list. Such a reference may now return stale data. Our solution to this problem is much like that of memory forwarding [19], except we can perform this optimization without processor modifications. Here, the coherence protocol implements the *safety net* by invalidating the original cache lines during the copying phase. If the processor accesses a dangling pointer, it is guaranteed to be a cache miss and can therefore be handled correctly by the active memory controller [15]. There are some limitations of this technique such as safety net overhead and potential pointer comparison problems [19], but it is still a powerful technique that shows large benefits in many applications.

2.4 Memory-side Merge

In the classic problem of parallel reduction we merge an array of elements by some operation which can be addition, multiplication, or even a maximum or minimum selector that can be used to sort an array of elements in memory. A final merge phase takes the locally reduced variables and reduces them to a single variable. Clearly, this merge phase suffers from remote read misses [4]. Our active memory merge can hide this miss latency and save the computation time of the merge phase. The technique is briefly explained below via dense matrix multiplication.

Suppose we want to compute $C = A^T B$ where A and B are dense matrices with compatible dimensions. To compute $C[i][j]$ we need to carry out an inner product of the i^{th} column of A and the j^{th} column of B. So, computation of each $C[i][j]$ is a parallel reduction with addition as the underlying operation. As shown in Figure 1, in a four-processor system the contribution of P_0 to $C[i][j]$ comes from carrying out the inner product of the shaded portions of the i^{th} column of A and j^{th} column of B. A final merge phase adds together all four parts of each $C[i][j]$ and generates the final result. This merge phase can be done completely in parallel by assigning each processor a range of mutually exclusive indices of C. Because this phase will incur many read misses while accessing the local sums, we do not carry out the merge phase on the main processor. Instead, we re-map the local sum arrays to the shadow space C', and whenever a shadow space cache line is written back to memory the memory controller adds its value to the corresponding cache line of C. With this optimization, the application does not have a merge phase and saves not only the read miss stall time but also the computation time of the merge phase. Further, if the writes to the merged array (in this case C) are sparse, the active memory controller saves some useless merge operations by never touching some cache lines [4].

In the memory-side merge, our controller maintains coherence between C and C'. If a cache line of the array C is accessed, it sends interventions for the corresponding lines belonging to the re-mapped spaces of local sums, performs addition on those local sums, returns the merged cache line and also writes the merged cache line back to main memory.

3. ACTIVE MEMORY CONTROLLER

In this section, we discuss the design goals and implementation of our active memory controller. We explain the microarchitecture in detail and illustrate the unique behavior of our controller via an example active memory transpose operation.

3.1 Design Goals

The design goals of our active memory controller are to provide flexibility in the types of active memory operations supported without sacrificing performance or changing the programming model. We achieve these goals by augmenting a programmable core, called the *Active Memory Processor Unit (AMPU)* with specialized hardware, called the *Active Memory Data Unit (AMDU)*. The AMPU runs software protocol handlers to implement cache coherence and control the correctness of active memory operations. The AMDU accelerates cache line assembly and disassembly, which form the datapath core of active memory techniques. By dividing our protocol execution into control and data paths (similar to the approach in [16]), and by executing them concurrently, we simultaneously achieve flexibility and performance.

3.1.1 Flexibility

In an all-hardware approach to an active memory controller, each active memory technique from Section 2 imposes its own specialized hardware requirements. Matrix transpose and sparse matrix scatter/gather need cache line assembly and disassembly capability, linked list linearization requires memory forwarding hardware and modifications to the main processor, and memory-side merge needs merging hardware at the memory controller. However, the basic underlying operations are the same—address re-mapping between the original data space and the shadow space, and word-granularity data operations. Therefore, a programmable active memory controller that can control hardware that efficiently supports these underlying operations can provide all of the active memory techniques above and more. In our active memory controller the AMPU runs the coherence protocol that controls the flow of data through the AMDU that supports both address re-mapping and word-size data operations. As the machine scales from a uniprocessor through single-node multiprocessors to multi-node multiprocessors, the software running on the AMPU is the only thing that needs to change to support active memory systems.

3.1.2 Performance

Active memory techniques reduce the cache miss rate and therefore the memory stall time. Some techniques, like memory-side merge, can also save busy time by performing computations at the memory controller. In conventional memory systems the key factor determining memory latency is the SDRAM access time. Although intelligent SDRAM page management policies can exploit the regularity in vector and stream accesses [21] and prediction techniques can improve SDRAM resource management [29], the raw SDRAM access latency still constitutes the major part of load-to-use latency in conventional memory systems. However, in active memory systems, the controller latency itself may become a bottleneck because of the overhead of active memory operations like address re-mapping. This issue is even more important in our architecture, because our controller performs not only active memory operations on the data, but also runs a cache coherence protocol to control those operations. We will see that we can achieve dramatic reductions in miss rate that more than compensate for the increase in latency during active memory operations. Further, the accesses to the normal address space are not affected by our active memory optimizations.

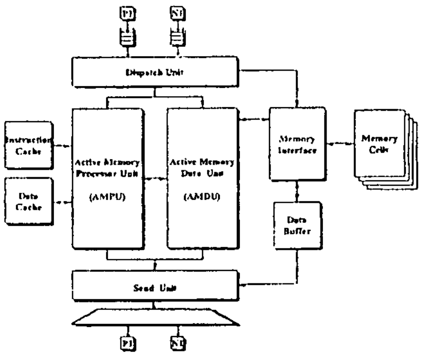

Figure 2: Controller Microarchitecture

To minimize the latency of active memory accesses, our specialized AMDU cache line assembly and disassembly engine supports fully-pipelined address calculation with the ability to issue a double word operation to the memory system on every system cycle. This functionality is important in the matrix transpose and sparse matrix active memory operations. The AMDU also has a dedicated adder to support memory-side merge, though the reduction operation can also be performed on the programmable AMPU. The AMPU latency must also be balanced with the AMDU so it does not become a bottleneck. A specialized ISA with instructions to tightly coordinate with the AMDU is a key to overlapping AMPU and AMDU operations to achieve high performance.

3.2 Microarchitecture

Figure 2 shows the architecture of our active memory controller. Memory requests are placed into one of two input queues (the network interface is used in multi-node systems) and are scheduled by the Dispatch Unit. The request is divided into header and data transfer components, which the AMPU and AMDU process concurrently. For active memory operations, the AMDU assembles or disassembles the cache line under the control of AMPU. Finally, the Send Unit returns the cache line to the requester if necessary. In the remainder of this section we describe each unit in detail.

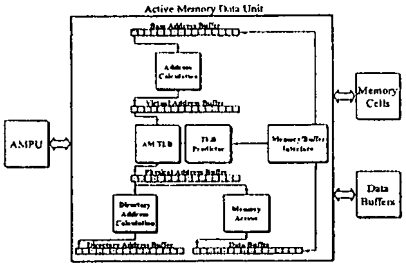

Figure 3: Active Memory Data Unit

Dispatch Unit. The Dispatch Unit schedules requests from the processor interface (PI) or network interface (NI) and initializes the AMPU and AMDU based on the address space and the type of the request.

Active Memory Processor Unit (AMPU). The AMPU is a dual-issue programmable core that executes the coherence protocol—the control portion of an active memory operation. It is a simple processor with a modified MIPS ISA. It does not support virtual memory, exceptions, floating-point arithmetic, or integer multiplication and division. However, it includes specialized instructions to enhance common cache coherence protocol and active memory operations. The AMPU gets its code and data from on-chip instruction and data caches, respectively. Both caches are backed by main memory.

For each memory request (normal or active), the AMPU executes the corresponding protocol handler. It checks and updates directory entries to preserve cache coherence, and sends appropriate control messages to the AMDU to perform active memory operations on data. The latency of the handler is critical to overall performance. For high performance the handler latency should be less than that of the AMDU so that it can be completely hidden by the data transfer time. In practice, we find that this is the case. As we show in Section 5.4, technology trends are also in our favor.

Active Memory Data Unit (AMDU). The AMDU (see Figure 3) is a specialized hardware datapath that performs pipelined address re-mapping and accelerates cache line assembly/disassembly. For each cache line, it loads/stores 16 different double words (a cache line) from/to the main memory according to the addresses it generates each cycle. We follow an idea similar to that proposed in [2]. However, because our cache coherence mechanism demands special operations from the AMDU, it shows quite different behavior, as discussed in Section 3.3.

The AMDU is composed of five cache line-sized buffers: *Base Address Buffer*, *Virtual Address Buffer*, *Physical Address Buffer*, *Directory Address Buffer*, and *Data Buffer*, and three pipelined stages: *Address Calculation*, *AMTLB Lookup*, and *Directory Address Calculation/Memory Access*. Each pipeline stage receives its input from the previous buffer and writes its result to the next buffer. Operations are fully pipelined, so one entry is processed per cycle. Therefore, the best-case latency of the AMDU is the pipeline latency + 15 cycles, where the pipeline latency is the time it takes one double word to pass through all three stages without stalls.

The Base Address Buffer contains technique-specific values that are intended to be used for virtual address calculation. For example, the sparse matrix technique uses this buffer to store *Acol*[*j*] values for the cache line under operation. The Address Calculation stage calculates virtual addresses from the Base Address Buffer by shift and add operations, and writes to the Virtual Address Buffer. Each entry of the Virtual Address Buffer holds the virtual address of the corresponding double word. The AMTLB (discussed below) translates the virtual addresses to physical addresses, and then the Memory Access stage performs a double word load/store operation to/from the corresponding entry of the Data Buffer. The Directory Address Calculation unit helps the AMPU calculate directory addresses. The cache coherence protocol requires the AMPU to check a directory entry for every double word, so the address calculation is performance-critical. By moving the address calculation to the AMDU, the latency of the AMPU handler is significantly reduced and because it operates concurrently with the Memory Access stage, it does not slow down the AMDU.

Active memory techniques directly manipulate application data that cannot be accessed through physical addresses. For example, linked list linearization traverses a list by chasing virtual addresses. The memory system is addressed with physical addresses, so the AMDU has a 256-entry direct-mapped TLB we call the AMTLB. Because an AMTLB miss stalls the AMDU pipeline and has a large miss penalty, the hit rate of the AMTLB is a critical determinant of performance. Therefore, our AMDU also has an AMTLB predictor to improve the performance of the AMTLB. The AMTLB predictor predicts and prefetches the next access to the AMTLB. It is a *Differential Finite Context Method* predictor [15, 27, 28, 7], which consists of 3 KB table and control logic. Detailed analysis of this predictor is given in Section 5.3. Note that although the AMTLB has 256 entries it is direct-mapped as opposed to the fully associative 64-entry data TLB of the processor. Finally, if the memory controller suffers a page fault, a trap is made to the kernel and the page fault handler is initiated.

The AMDU is under full control of the AMPU, though both units run concurrently. The AMPU sets parameters such as the shift amount in the Address Calculation stage, and it can read and write all the buffers.

Send Unit. The Send Unit is responsible for the mechanics of sending intervention or reply messages sent by the AMPU. The Send Unit inserts the message into the corresponding output queue (PI or NI), offloading this task from the AMPU. The Dispatch Unit, AMPU, and Send Unit can all operate concurrently on different requests.

Memory Interface. The Memory Interface connects the SDRAM to the other parts of the controller. It picks a request from a 16-deep request queue and performs loads or stores. It is fully pipelined and the AMDU does not stall unless the memory request queue fills.

3.3 Example: Matrix Transpose

We present the matrix transpose operation as an example to illustrate the features of our active memory controller. Assume that A is a square matrix of dimension N and A' is a shadow space mapped to A by our matrix transpose technique. Note that our programming model allows accesses to the normal space A and the shadow space A' without the need to worry about coherence between the two.

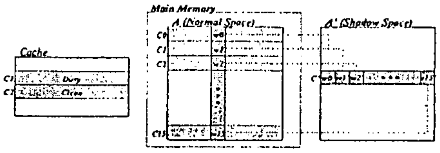

Figure 4: Example: Matrix Transpose

```
A' = AMInstall(A, N, N, sizeof(Complex));
Initialize(A);
for i=0 to N-1
  for j=0 to N-1
    x += A'[i][j];
for i=0 to N-1
  for j=0 to N-1
    x += A[i][j];
```

Initialization. The transpose application starts by invoking a setup library call `AMInstall` that passes some basic information to the active memory controller—the virtual address of A, the dimension N, and the size of each element of the matrix. The active memory controller stores the information in its data structures and returns the shadow address A' mapped to A. It also stores the virtual address of the shadow space.

Forward Mapping. The first loop accesses A' in row-major fashion. Assume the situation depicted in Figure 4. The processor reads a cache line C' of A' that is clean in memory. C' is composed of 16 double words that map to $w0, w1, \ldots, w15$, and the 16 cache lines $C0, C1, \ldots, C15$ of A contain these double words. The processor may be caching one or more of $C0$ to $C15$ when it accesses C'. In this example, let us assume the processor cache has $C1$ and $C2$ in the dirty and shared states, respectively, and that the other lines are clean in memory.

When the processor reads C' it is a cache miss and the processor sends a memory request to the active memory controller. The message is inserted into the PI queue, and the Dispatch Unit schedules it, checks the address space of the request, and initializes the AMDU accordingly. The AMPU is instructed to run the matrix transpose protocol handler and the AMDU starts cache line assembly. While the AMPU checks the directory entries of $C0, C1, \ldots, C15$, the AMDU concurrently assembles the cache line C'. The AMDU speculatively assumes that every double word required is clean in memory. The Address Calculation stage calculates the virtual Addresses of $w0, w1, \ldots, w15$, which are translated to physical addresses by the AMTLB. The Memory Access stage requests 16 double word reads from the Memory Interface. After the initial memory access latency, a double word is fed into the Data Buffer once per cycle.

Meanwhile, the AMPU reads the directory entries of $C0$, $C1, \ldots, C15$ and finds that $C1$ and $C2$ are cached in the dirty and shared states, respectively. The AMPU sends an intervention to the main processor for $C1$. In the AMDU, the Data Buffer now has a stale value for $w1$ because the most recent data is in the processor cache. On receiving the intervention reply the AMPU writes $C1$ to the memory and issues a control message instructing the AMDU to get the correct data for $w1$. The AMPU also sends an invalidation for $C2$, but does not issue a control message to the AMDU, because it already has the correct data in this case. Finally, the AMPU and AMDU finish their work and the Data Buffer has an assembled cache line C'. The AMPU issues a send command to the Send Unit, and the Send Unit sends C' to the processor. Cache coherence plays an important role to guarantee correctness in this example. The active memory controller sends an intervention for $C1$ and an invalidation for $C2$, before it returns C' to the main processor, guaranteeing mutual exclusion between C' and $C0, C1, \ldots, C15$.

In this example, the AMDU performs cache line assembly. A similar cache line disassembly takes place when the memory controller receives a writeback for a shadow cache line. To generate the addresses for the 16 double words, the AMDU calculates a *forward mapping* from A' to A as follows. First, from the physical address of C', the row and column indices of C' are calculated with the help of the information provided by the initialization call. Second, the indices are transposed. Third, the virtual addresses of $w0, w1, \ldots, w15$ are calculated from the indices and the starting virtual address of A. Finally, the AMTLB translates the virtual addresses into physical addresses.

Inverse Mapping. The corresponding inverse mapping from A to A' is needed when the second loop accesses A. From the physical address of C, the physical addresses of $w0', w1', \ldots, w15'$ are calculated, where $w0', w1', \ldots, w15'$ are the corresponding double words of A'. Cache coherence guarantees correctness in the same fashion as before. The AMPU checks the directory entries of $C1', C2', \ldots, C15'$, which are the cache lines containing $w0', w1', \ldots, w15'$, respectively. For dirty cache lines, it sends an intervention and writes back the replied cache line to memory, and it sends invalidations for shared cache lines.

4. APPLICATIONS AND SIMULATION METHODOLOGY

In this section we discuss the steps necessary to convert a normal application into an active memory application, the applications we use to evaluate the performance of our active memory system, and the simulation environment we use to collect the results.

4.1 Programmer Implications

To exploit the flexibility of the active memory controller an application programmer needs to follow a few simple steps that can be easily automated with a compiler for most applications, though that is not our focus here. First, we identify an active memory operation in the application that the system supports and the data structures where the results of this operation are stored when the operation is applied. We will collectively refer to these data structures as R. Next, we allocate a virtual address space of size R and map it to our physical shadow address space. Recall that the shadow address space does not exist in the physical memory but is used only to help the memory controller distinguish active memory accesses from normal memory accesses. At the beginning of the application we insert an initialization library call to set up a table of values that the flexible controller uses while carrying out the active memory operations as previously described in Section 3.3. Then, since the active memory operations will be performed by the active memory controller, we remove all instances of the operation from the application code. In the original code all accesses to R after the active memory operation are replaced by a corresponding access to the shadow address space. A detailed example can be found in our technical report [15].

Table 1: Applications and Problem Sizes

Applications	Problem Sizes
SPLASH-2 FFT	1M(1K×1K) complex doubles
FFTW	2M(8K×16×16) complex doubles
Transpose	1M(1K×1K) complex doubles
Conjugate Gradient	8K×8K matrix, 256K non-zeros
SMVM	64K×64K matrix, 2M non-zeros
MST	2K node graph
Health	6 level tree, 4 children per node
Traverse	256 lists, 1K elements per list
MMM	64K(256×256) elements
SparseFlow	64K nodes, 8K edges
Parallel Reduction	512K elements

4.2 Applications

To evaluate each of the techniques discussed in Section 2 we use a range of applications—some are well-known benchmarks while others are microbenchmarks written to exhibit the potential of a particular active memory technique. We use FFT from SPLASH-2 [35], FFTW [3], and a microbenchmark called Transpose to evaluate the performance of the matrix transpose active memory technique. The microbenchmark reads and writes to an array and its transpose. As sparse matrix applications we use Conjugate Gradient from the DIS (Data-Intensive Systems) benchmark suite [33] and a microbenchmark called SMVM that carries out the sparse matrix vector multiplication kernel. Linked list linearization is evaluated by running MST [1] and Health [1] from the Olden benchmarks, and a microbenchmark called Traverse that walks through a number of lists as new elements are inserted. The length of each list increases to a maximum of 1024 nodes. Finally, to evaluate parallel reduction we use the dense matrix multiplication (MMM) described in Section 2.4, a microbenchmark called SparseFlow that carries out some operation on the in-flow of each node and sums them in a sparse multi-source flow graph, and a microbenchmark called Reduction that performs a parallel reduction on an array of elements. In Table 1 we summarize the applications and the problem sizes we use in simulation.

4.3 Simulation Methodology

In Section 5 we present simulation results of the applications above for four different cases: the normal application, the application running with our active memory support, the application running with active memory support and software prefetching for shadow address space access only, and the application running with active memory operations but relying on cache flushing rather than coherence. Our simulator models contention in detail within the active memory controller, between the controller and its external interfaces, at main memory, and for the system bus. The embedded active memory processor is a dual-issue core running at the 400 MHz system clock frequency, and executing the code sequences that comprise our coherence handlers. We simulate an invalidation-based bitvector protocol running under release consistency. Each directory entry is byte-sized with four bits dedicated to the sharer list, one bit to mark whether the cache line is re-mapped or not, one bit to mark if the the line is dirty and two bits are left unused. The instruction and data cache behavior of the active memory processor is modeled precisely via a cycle-accurate simulator similar to that for the protocol processor in [6]. The input and output queue sizes in the memory controller's processor interface are set at 16 and 2 entries respectively. We assume processor interface delays of 1 system cycle inbound and 4 system cycles outbound. The access time of main memory SDRAM is fixed at 125 ns (50 system cycles), similar to that in recent commercial high-end servers [30, 32].

The main processor runs at 2 GHz and is equipped with separate 32 KB primary instruction and data caches that are two-way set associative and have a line size of 64 bytes. The secondary cache is unified, 512 KB in size, two-way set associative, and has a line size of 128 bytes. For sparse matrix applications we scale down the cache size so that we can simulate the effect of running problems with large sparse matrices by running smaller problem sizes that we can simulate within a reasonable amount of time. For this class of applications we use a 16 KB primary data cache and a 64 KB secondary cache keeping the same line sizes and associativities. We also assume that the processor ISA includes prefetch and prefetch exclusive instructions. In our processor model a load miss stalls the processor until the first double-word of data is returned, while prefetch and store misses will not stall the processor unless there are already references outstanding to four different cache lines. The processor model also contains fully-associative 64-entry instruction and data TLBs and we accurately model the latency and cache effects of TLB misses.

To minimize the flush overhead we simulate user-level complete cache flushes. This does not involve any kernel trap overhead, but it does model the latency incurred in the cache hierarchy to flush the whole cache. Note that systems that only support selective page flushes will see a larger flush overhead because realistic problem sizes are far bigger than the cache size and the entire data structure is flushed one page at a time.

5. SIMULATION RESULTS

Our simulation results are broadly divided into four areas: uniprocessor active memory systems, single-node multiprocessor active memory systems, a study of the performance of our AMTLB predictor, and the effects of continued technology scaling on our active memory architecture.

5.1 Uniprocessor Active Memory Systems

We report uniprocessor results for three active memory operations: matrix transpose, sparse matrix and linked list linearization. For the first two techniques we present speedup of the active memory version over the normal application, the speedup of the active memory version with software prefetching of the shadow address space (where prefetching is not possible in the normal application as explained in Section 2), and the speedup of active memory applications using cache flushes rather than coherence. For applications involving linked list linearization, cache flush results have no relevance and are not shown since this technique requires leveraging the coherence mechanism. We note that although non-active applications are run with the same flexible active memory controller, this does not affect normal uniprocessor execution time since protocol processing is minimal and the memory access time completely dominates the AMPU handler latency as in [9].

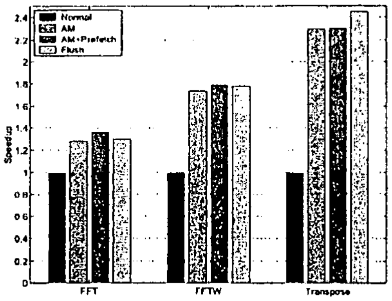

Figure 5: Uniprocessor Speedup: Matrix Transpose

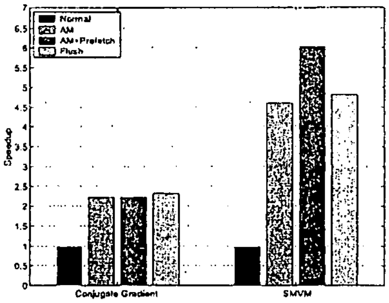

Figure 6: Uniprocessor Speedup: Sparase Matrix

5.1.1 Matrix Transpose

Figure 5 shows the uniprocessor speedup of FFT, FFTW, and the Transpose microbenchmark with active memory optimization (AM), with active memory and software prefetching of the shadow address space (AM+Prefetch), and with active memory using cache flush calls rather than coherence (Flush), measured relative to the execution time of the normal application. All the applications show the clear success of the matrix transpose operation in our active memory system. FFT with active memory optimization runs 1.28 times faster than the normal application and with prefetching in the transformed shadow address space it is 1.36 times faster. While both the normal and active memory executions could benefit equally from prefetching row-wise accesses in the normal address space, here we emphasize that the prefetches from the shadow address space are a bonus of the active memory technique. The active memory speedup is mainly due to a factor of 2.0 reduction in L2 cache read misses and a reduction of the overall processor data TLB miss penalty by a factor of 176. The speedup for FFTW is even larger than FFT, achieving 1.74 with active memory optimization and 1.78 with software prefetching. For this application we reduce the overall L2 cache read misses by a factor of 4.0 and the overall processor data TLB miss penalty by a factor of 53.8. The Transpose microbenchmark is a highly memory-bound application that reads and writes to the normal matrix and its transpose, but performs little computation. It achieves a speedup of 2.3 over normal execution. Because of the lack of computation, it is difficult for prefetching to hide memory latency in this microbenchmark and there is little additional benefit from prefetching. Active memory optimization for this microbenchmark reduces L2 cache read misses by a factor of 3.8 and the overall processor data TLB miss penalty by a factor of 72.

For FFT and FFTW flush has marginally better performance than our coherence-based active memory system. Given the advantages that a coherence-based solution brings to the programming model along with multiprocessor correctness, we note that our results show that it is possible for a coherence-based approach to achieve these advantages at a performance level commensurate with the cache flush technique.

5.1.2 Sparse Matrix

Figure 6 shows the uniprocessor speedup for Conjugate Gradient and the Sparse Matrix Vector Multiply (SMVM) microbenchmark. As described in Table 1, both applications have relatively sparse matrices—32 non-zeros on average per row. For Conjugate Gradient the dense vector has length 8192 and fits into the scaled-down 64 KB L2 cache we use for these sparse matrix simulations. CG therefore represents a moderately large problem size while the SMVM microbenchmark shows results for a much bigger problem size. Conjugate Gradient achieves a uniprocessor speedup of 2.22 while SMVM is 4.62 times faster than the normal application. With software prefetching of the shadow address space the speedup increases to 2.23 and 6.04 respectively. The SMVM kernel is particularly well-suited to this optimization. For Conjugate Gradient, active memory optimization reduces L2 cache read misses by a factor of 3.8 while for SMVM the reduction factor was 7.8. For these two applications the reduction factors in the overall processor data TLB miss penalty were 2.1 and 94.2 respectively. For sparse matrix applications we see that the flush technique is marginally better than coherence. However, since we scale down the caches to simulate the effect of large sparse matrices, and in our flushing scheme the flush overhead depends on the size of the cache rather than the size of the matrix, the flush scheme receives some relative benefit in the results presented here.

5.1.3 Linked List Linearization

Figure 7 shows the uniprocessor speedup for Health, MST, and the Traverse microbenchmark. Health achieves a speedup of 1.31 with linearization, increasing to 1.37 with software prefetching. In Health every node in the tree has several linked lists attached to it, but we linearize only two of them to demonstrate the potential of this technique. The remaining lists can be linearized in a similar manner to achieve better speedup. For Health the linearization technique reduces the number of L2 cache read misses by a factor of 1.3. MST represents a complete graph of 2048 nodes as a hash table. Since hashing collisions are resolved by chaining, the bigger the hash table the smaller the average length of each linked list. As expected, the linearization technique gets more benefit from a longer linked list. We use a hash table size of $N/32$ where N is the number of nodes in the graph. Also, MST has both a graph-building phase where we linearize the $N/32$ lists as each list grows, and a compute phase that calculates the minimum spanning tree. Figure 7 shows the overall speedup for MST by including both phases. The overall speedup is 2.28 without prefetching and increases to 2.53 with prefetching. The speedup of the computation phase only (not shown) is 3.74 without prefetching and 4.56

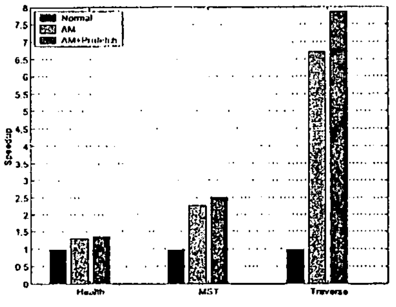

Figure 7: Uniprocessor Speedup: Linearization

with prefetching, but this does not account for the linearization overhead and so we show total speedup here. Including the linearization overhead, the linearization technique reduces L2 cache read misses by a factor of 4.1. Traverse shows even better speedup because of a larger number of longer lists. It achieves a speedup of 6.72 without prefetching with a 6.6 times reduction in L2 cache read misses. The speedup increases to 7.68 with software prefetching. We would again like to emphasize that prefetching is only possible because of the linearization technique, and is not possible in the normal application. In addition, linked list linearization is only possible using a coherence-based approach, hence no flush comparisons can be shown.

5.2 Single-node Multiprocessor Active Memory Systems

Since our active memory system leverages the cache coherence protocol, it can transparently support address remapping techniques on multiprocessor systems. In the following we present single-node multiprocessor results for an active memory transpose version of SPLASH-2 FFT and three applications for Parallel Reduction. We show results for one, two, and four processor systems.

5.2.1 Fast Fourier Transform

Figure 8 shows the single-node multiprocessor speedup for FFT, calculated relative to the uniprocessor execution time of the normal application. For all the processor counts, FFT with an active memory transpose beats the normal application. The dual-processor execution with in-memory transpose achieves a speedup of 2.42 while the quad-processor node achieves a speedup of 3.84. Stated differently, for a dual-processor node, in-memory transpose with cache coherence is 1.26 times faster than the normal two-processor execution. For a quad-processor node the corresponding speedup is only 1.13, due to increased AMPU occupancy. The avergge AMPU occupancy for 1, 2 and 4 processors in the normal executions is 7.7%, 21.3% and 43.2% of the total execution time, respectively. The corresponding occupancy for the active memory executions is 15.5%, 29.4% and 47.8%, though one must remember that the active memory execution time is smaller. These results show that our coherence-based active memory techniques scale to multiprocessor nodes. We discuss techniques to further reduce AMPU occupancy in Section 6.

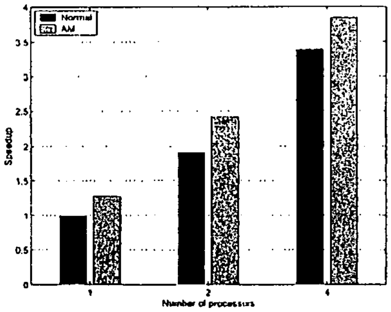

Figure 8: Multiprocessor Speedup: FFT

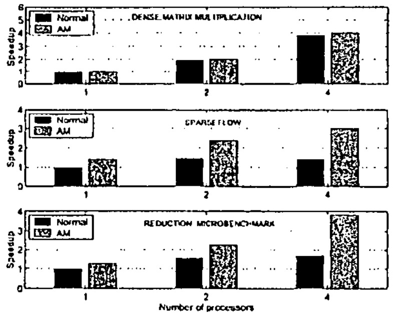

Figure 9: Multiprocessor Speedup: Reduction

5.2.2 Parallel Reduction

Figure 9 shows the single-node multiprocessor speedup for three applications that have parallel reduction kernels. As previously mentioned, the main reason for using memory-side merge is to hide remote miss latency. But in a single-node multiprocessor every cache miss results in a local memory access. Though we expect even larger gains from this technique in multi-node systems, we can still show improvements in a single-node system. Because the merge phase in a single-node system may cause many cache interventions, a single-node multiprocessor active memory system can save busy time and the cost of those interventions. For dense matrix multiplication both normal and active memory applications scale well on a single node, with the active memory technique being only slightly better. This is because the computation time involved in the parallel matrix multiplication phase is much bigger than the time spent in the merge phase, especially in the absence of significant network latency. For SparseFlow and the Reduction microbenchmark one can clearly see saturating trends as the normal application scales, while the active memory technique scales well. In SparseFlow our active memory technique benefits from saving useless merges that happen in the normal application for sparse data structures. The Reduction microbenchmark also achieves good speedup because of well-balanced computation and merge phases. With the network latencies of multi-node systems, we expect that the merge phase will always dominate the local computation phase and this active memory technique will be an even bigger win. We discuss this and other future research in Section 6.

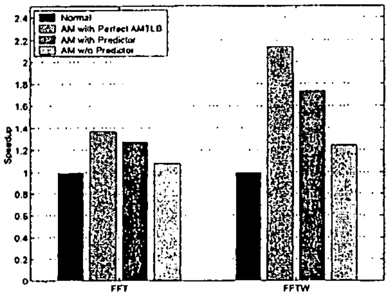

Figure 10: 256-entry Direct Mapped AMTLB

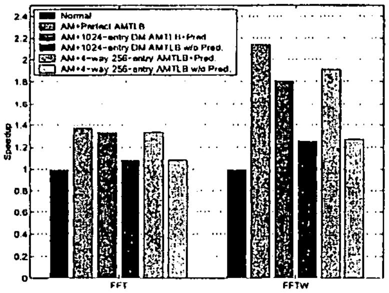

Figure 11: Variation of Entries and Set Associativity

5.3 AMTLB Predictor Study

In our simulations we found that the active memory matrix transpose suffers from high AMTLB miss rates. FFT and FFTW show 28.7% and 33.5% miss rates, respectively [15]. To overcome this problem, our controller uses an AMTLB predictor. Here, we study various aspects of our predictor.

Figure 10 shows the speedup of FFT and FFTW with different AMTLB configurations. With a perfect AMTLB every access is a hit. Both applications show that the AMTLB predictor is quite effective. We found that the predictor reduces the AMTLB miss rate from 28.7% to 7.3% for FFT and from 33.5% to 10.6% for FFTW, so that FFT is 19% and FFTW is 40% faster compared to the execution without an AMTLB predictor. Figure 11 shows the speedup with a bigger AMTLB (1024 entries) and with a set-associative AMTLB of the same size (256 entries and 4-way). The results show that increasing the AMTLB size does not improve performance without a predictor because the data sizes of FFT and FFTW are bigger than the coverage of the AMTLB and the data access pattern keeps the miss rate the same. Associativity only marginally helps FFTW, with a 6% speedup over the 1024-entry direct-mapped AMTLB. When using our predictor with the larger AMTLB configurations we found that the speedup of FFTW increased by only 4% over the 256-entry direct-mapped AMTLB with the predictor. We also carried out a comparison between a 256-entry direct-mapped AMTLB with predictor and a larger direct-mapped AMTLB with no predictor but of equal size to the total size of the smaller AMTLB with the predictor. We found that for both applications the smaller AMTLB with our predictor dramatically outperformed the larger AMTLB with no predictor, by 16% and 33% respectively.

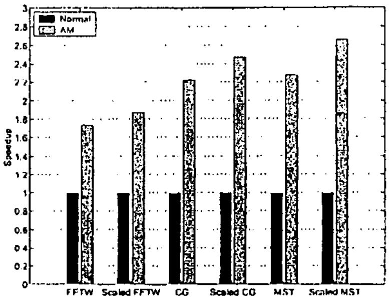

Figure 12: Effects of Technology Scaling

5.4 Effects of Technology Scaling

Logic speeds continue to outpace memory access time as technology scales. Just as the main processor speed increases, so does the frequency of our embedded active memory processor unit (AMPU). However, raw SDRAM access times improve much more slowly. Figure 12 summarizes the results for two different technologies—one is our base technology with a 2 GHz processor, 400 MHz memory system, and 125 ns SDRAM access time; the other is a scaled technology where both the processor and the AMPU are four times faster compared to the base technology while keeping the SDRAM access time unchanged. The results are presented for three applications: FFTW, Conjugate Gradient and MST. For a particular technology and a particular application the speedup is shown relative to the normal execution time for that technology and application. For all the applications our approach gracefully scales with future technology with even better speedup. As the gap between the processor speed and the SDRAM access time widens, active memory techniques will show even larger performance improvements.

6. FUTURE WORK

We have shown significant speedup on uniprocessor and single-node multiprocessor applications for four active memory operations. We continue to look for ways to exploit the flexibility of our active memory controller. Possibilities include implementing memory-side prefetch techniques that do not require any application modification [37]. On multiprocessor systems this will once again require a coherence-based approach to active memory. We will also investigate the inclusion of active memory elements (instead of standard SDRAM) to form what we call two-level active memory systems [20] where the active memory controller manages coherence and the active memory elements perform data parallel operations.

Our active memory architecture also contains all the necessary functionality to support coherent multi-node systems. With the evolution of system area networks like 3GIO [12] and InfiniBand [11] that are integrated at the memory controller, it is possible to form cache-coherent clusters with the same active memory controller that provides performance benefits on uniprocessors and single-node multiprocessors. We call such a system *active memory clusters* [10]. To support active memory clusters, the only necessary addition to our system is the software coherence handlers that handle network requests. This does not necessitate any hardware changes to our design, though to scale to larger clusters we may explore the use of multiple embedded cores as in [23, 24]. The flexibility of active memory clusters will also let us explore the synergy between active memory operations and traditional multiprocessing functions (e.g. we would expect larger gains from our parallel reduction operation), as well as exploring coherence protocols that are efficient on single-node systems yet scale well to larger coherent clusters. We can also explore predictive techniques in these scalable systems similar to those in [17, 22].

7. CONCLUSIONS

Our active memory architecture improves the performance of uniprocessor and multiprocessor systems when they exhibit poor cache behavior. In this paper, we have detailed the microarchitecture of a flexible active memory controller that extends the cache coherence mechanism to implement active memory operations without requiring cache flushes by the programmer. We described four active memory operations that perform address re-mapping techniques to improve spatial locality and reduce the number of cache misses in both uniprocessor and single-node multiprocessors. The address re-mapping creates a coherence problem that our active memory controller solves by enforcing mutual exclusion between the caching states of the two spaces, providing a transparent and safe programming model to extend traditional uniprocessor active memory techniques to multiprocessor systems.

Through detailed simulation on a range of applications we have shown that our active memory system achieves uniprocessor speedup from 1.3 to 7.6. We have also shown that these impressive speedup numbers can be improved by software prefetching the shadow address space where our active memory transformations have created spatial locality that was not present in the original code. Further, we have shown that the architecture scales to a single-node multiprocessor system and can improve the speedup of parallel active memory applications as well.

In addition to transparency, another focus of this work is the flexibility of the active memory controller at the heart of the system. The flexibility allows us to run both traditional active memory operations, such as in-memory matrix transpose and sparse matrix scatter/gather operations, and non-traditional active memory operations like linked list linearization, parallel FFT, and parallel reduction. Customized instructions in our embedded processor and a highly-optimized data unit that performs pipelined cache line assembly and disassembly strike a balance between flexibility and performance. We have also introduced an AMTLB predictor that improves the speedup of active memory operations by up to 40%.

The transparency and flexibility of our system make it possible to extend our approach to multi-node systems with active memory support called active memory clusters. We are beginning to look at the intriguing possibilities of such systems, building on the scalable coherent single-node architecture and well-understood programming model described here. As processor performance continues to outpace the memory system, active memory architectures become increasingly attractive, especially on multi-node shared memory systems where remote memory latencies can be quite large.

ACKNOWLEDGMENTS

This research was supported by Cornell's Intelligent Information Systems Institute and NSF CAREER Award CCR-9984314.

REFERENCES

[1] M. C. Carlisle and A. Rogers. Software Caching and Computation Migration in Olden. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 29–38, July 1995.

[2] J. B. Carter et al. Impulse: Building a Smarter Memory Controller. In *Proceedings of the Fifth International Symposium on High Performance Computer Architecture*, January 1999.

[3] M. Frigo and S. G. Johnson. FFTW: An Adaptive Software Architecture for the FFT. In *Proceedings of the 23rd International Conference on Acoustics, Speech, and Signal Processing*, pages 1381–1384, 1998.

[4] M. J. Garzaran et al. Architectural Support for Parallel Reductions in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 10th International Conference on Parallel Architectures and Compilation Techniques*, September 2001.

[5] K. Gharachorloo et al. Architecture and Design of AlphaServer GS320. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 13–24, November 2000.

[6] J. Gibson et al. FLASH vs. (Simulated) FLASH: Closing the Simulation Loop. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 49–58, November 2000.

[7] B. Goeman, H. Vandierendonck, and K. D. Bosschere. Differential FCM: Increasing value prediction accuracy by improving table usage efficiency. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, January 2001.

[8] M. Hall et al. Mapping Irregular Applications to DIVA, A PIM-based Data-Intensive Architecture. In *Proceedings of Supercomputing*, November 1999.

[9] M. Heinrich et al. The Performance Impact of Flexibility in the Stanford FLASH Multiprocessor. In *Proceedings of the sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 274–285, October 1994.

[10] M. Heinrich, E. Speight, and M. Chaudhuri. Active Memory Clusters: Efficient Multiprocessing on Commodity Clusters. In *Proceedings of the Fourth International Symposium on High-Performance Computing*, May 2002.

[11] InfiniBand Trade Association. *InfiniBand Architecture Specification, Volume 1.0, Release 1.0*, October 2000.

[12] Intel, http://developer.intel.com/technology/3gio/. *Creating a Third Generation I/O Interconnect.*

[13] Y. Kang et al. FlexRAM: Toward an Advanced Intelligent Memory System. In *Proceedings of International Conference on Computer Design*, October 1999.

[14] D. Keen et al. Cache Coherence in Intelligent Memory Systems. In *ISCA 2000 Solving the Memory Wall Problem Workshop*, June 2000.

[15] D. Kim, M. Chaudhuri, and M. Heinrich. Leveraging Cache Coherence in Active Memory Systems. Technical Report CSL-TR-2001-1018, Computer Systems Laboratory, Cornell University, November 2001.

[16] J. Kuskin et al. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 302–313, April 1994.

[17] A.-C. Lai and B. Falsafi. Memory Sharing Predictor: The Key to a Speculative Coherent DSM. In *Proceedings of the 26th International Symposium on Computer Architecture*, pages 172–183, May 1999.

[18] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 241–251, June 1997.

[19] C.-K. Luk and T. C. Mowry. Memory Forwarding: Enabling Aggressive Layout Optimizations by Guaranteeing the Safety of Data Relocation. In *Proceedings of the 26th International Symposium on Computer Architecture*, pages 88–99, May 1999.

[20] R. Manohar and M. Heinrich. A Case for Asynchronous Active Memories. In *ISCA 2000 Solving the Memory Wall Problem Workshop*, June 2000.

[21] B. K. Mathew et al. Algorithmic Foundation for a Parallel Vector Access Memory System. In *Proceedings of the 12th ACM Symposium on Parallel Algorithms and Architectures*, pages 156–165, July 2000.

[22] S. S. Mukherjee and M. D. Hill. Using Prediction to Accelerate Coherence Protocols. In *Proceedings of the 25th International Symposium on Computer Architecture*, pages 179–190, 1998.

[23] A. K. Nanda et al. High-Throughput Coherence Controllers. In *Proceedings of the Sixth International Symposium on High-Performance Computer Architecture*, January 2000.

[24] A. Nowatzyk et al. The S3.mp Scalable Shared Memory Multiprocessor. In *Proceedings of the 24th International Conference on Parallel Processing*, 1995.

[25] M. Oskin, F. T. Chong, and T. Sherwood. Active Pages: A Computation Model for Intelligent Memory. In *Proceedings of the 25th International Symposium on Computer Architecture*, 1998.

[26] A. Saulsbury, F. Pong, and A. Nowatzyk. Missing the Memory Wall: The Case for Processor/Memory Integration. In *Proceedings of the 23rd International Symposium on Computer Architecture*, pages 90–101, May 1996.

[27] Y. Sazeides and J. E. Smith. Implementations of Context Based Value Predictors. Technical Report ECE-97-8, University of Wisconsin-Madison, December 1997.

[28] Y. Sazeides and J. E. Smith. The Predictability of Data Values. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, December 1997.

[29] R. Schumann. Design of the 21174 Memory Controller for DIGITAL Personal Workstations. *Digital Technical Journal*, 9(2), January 1997.

[30] Silicon Graphics, http://www.sgi.com/origin/3000/. *SGI 3000 Family Reference Guide.*

[31] Y. Solihin, J. Lee, and J. Torrellas. Adaptatively Mapping Code in an Intelligent Memory Architecture. In *Proceedings of the Second Workshop on Intelligent Memory Systems*, November 2000.

[32] Sun Microsystems, http://www.sun.com/servers/white-papers/. *Sun Enterprise 10000 Server–Technical White Paper.*

[33] Titan Systems, http://www.aaec.com/projectweb/dis/. *DIS Benchmark Suite.*

[34] J. Torrellas, L. Yang, and A. T. Nguyen. Toward a Cost-Effective DSM Organization that Exploits Processor-Memory Integration. In *Proceedings of the Sixth International Symposium on High-Performance Computer Architecture*, pages 15–25, January 2000.

[35] S. C. Woo et al. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, June 1995.

[36] L. Zhang et al. Memory System Support for Dynamic Cacheline Assembly. In *Proceedings of the Second Workshop on Intelligent Memory Systems*, November 2000.

[37] L. Zhang et al. Pointer-Based Prefetching within the Impulse Adaptable Memory Controller: Initial Results. In *Proceedings of the ISCA-2000 Workshop on Solving the Memory Wall Problem*, June 2000.

Cache Coherence Protocol Design for Active Memory Systems

Mainak Chaudhuri, Daehyun Kim, and Mark Heinrich
Computer Systems Laboratory, Cornell University, Ithaca, NY, U.S.A.

Abstract— *Active memory systems improve application cache behavior by either performing data parallel computation in the memory elements or supporting address re-mapping in a specialized memory controller. The former approach allows more than one memory element to operate on the same data, while the latter allows the processor to access the same data via more than one address—therefore data coherence is essential for correctness and transparency in active memory systems. In this paper we show that it is possible to extend a conventional DSM coherence protocol to handle this problem efficiently and transparently on uniprocessor as well as multiprocessor active memory systems. With a specialized programmable memory controller we can support several active memory operations with simple coherence protocol code modifications, and no hardware changes. This paper presents details of the DSM cache coherence protocol extensions that allow speedup from 1.3 to 7.6 over normal memory systems on a range of simulated uniprocessor and multiprocessor active memory applications.*

Keywords: Active memory systems, address re-mapping, cache coherence, distributed shared memory, flexible memory controller.

1 Introduction

Active memory systems provide a promising approach to overcoming the memory wall for applications with irregular access patterns not amenable to techniques like prefetching or improvements in the cache hierarchy. The central idea in this approach is to perform data-parallel computations [2, 4, 7] or scatter/gather operations invoked via address remapping techniques [1] in the memory system to either offload computation directly or to reduce the number of processor cache misses. Both active memory approaches create coherence problems—even on uniprocessor systems—since there are either additional processors in the memory system operating on the data directly, or the main processor is allowed to refer to the same data via more than one address.

This paper focuses on the challenges of designing cache coherence protocols for active memory systems. The necessity of enforcing data coherence between the normal cache line and the re-mapped cache line makes the protocol behavior and the performance requirements quite different from those of a conventional DSM protocol. We propose an active memory system that supports address remapping by leveraging and extending the hardware DSM directory-based cache coherence protocol. The key to our approach is that the active memory controller not only performs the remapping operations required, but also runs the directory-based coherence protocol and hence controls which mappings are present in the processor caches.

The paper is organized as follows. Section 2 describes the protocol extensions required and protocol implementation issues unique to active memory systems. Section 3 discusses active memory protocol-related performance issues. Section 4 presents both uniprocessor speedup and the performance improvement of active memory applications on single-node multiprocessors. Section 5 concludes the paper.

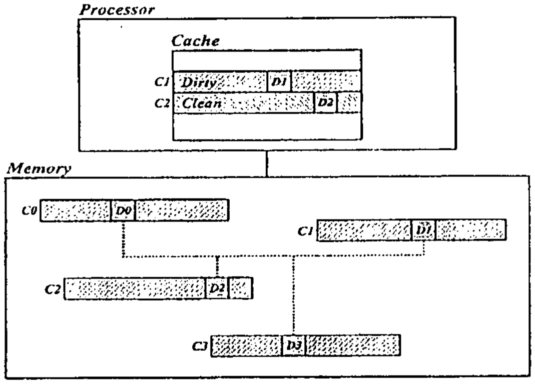

Figure 1. Data Coherence Problem

2 DSM Protocol Extensions for Active Memory Systems

Our embedded memory controller runs software code sequences to implement the coherence protocol following the same philosophy as the FLASH multiprocessor [6]. However, the controller also includes specialized hardware to speed up active memory protocol execution. Our base (non-extended) protocol is a conventional invalidation-based MSI bitvector directory protocol running under release consistency. For all normal (non-re-mapped) memory requests, our memory controller follows this base protocol. Each directory entry (per 128B cache line) is 8 bits wide. The sharer vector occupies 4 bits, so we can support up to 4 processors. This can be expanded to larger machine sizes by increasing the width of the sharer field. These four bits store a sharer list vector if the cache line is in the shared state or an owner identifier for the dirty exclusive state. Two bits are devoted to maintain cache line state information. The dirty bit indicates whether a cache line is in the dirty exclusive state. The AM bit is used for our active memory protocol extensions and is not used by the base protocol. Two remaining bits are left unused.

2.1 Active Memory Extensions

As an example of address remapping techniques giving rise to a data coherence problem, Figure 1 shows that the data element $D0$ in cache line $C0$ is mapped by some address remapping technique to data elements $D1$, $D2$ and $D3$ belonging to three different cache lines $C1, C2$ and $C3$, respectively. This means that $D1, D2$ and $D3$ represent the same data variable as $D0$ and our active memory protocol extensions must keep them coherent. As in the figure, if $C1$ and $C2$ are cached in the dirty and shared state, respectively, the processor may write a new value to $D1$, but read a stale value from $D2$. We extend the base cache coherence protocol to enforce mutual exclusion between the caching states of the lines that are mapped to each other so that only one cache line of the four mapped cache lines (as in the example above) can be cached at a time. If the processor accesses another mapped cache line it will suffer a cache miss and our protocol will invalidate the old cache line before replying with the requested cache line.

For each memory request, our protocol consults the AM bit in the directory entry for the requested cache line. The AM bit of a cache line indicates whether any data in the line has been re-mapped and is being cached by processors in the system at a different address. If the AM bit is clear, there is no possibility of a coherence violation and the protocol behaves just like the base protocol with one additional task—to guarantee coherence in the future, the protocol sets the AM bits in the directory entries of all the cache lines that are mapped to the requested line. However, if the AM bit is set, some of the re-mapped cache lines (we collectively call these lines R) are cached in the system and there is a potential data coherence problem. The caching states of R are obtained by reading the correspond-

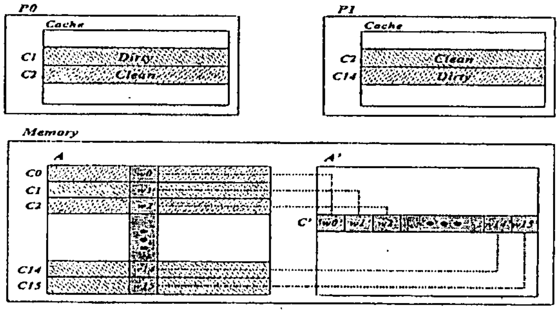

Figure 2. Example - Matrix Transpose

ing directory entries. If R is in the dirty exclusive state, the protocol sends an intervention to the owner of R to retrieve the most recent copy of the data, updates the mapped data value in the requested cache line, sends the data reply to the requester, and writes back the retrieved cache line to main memory. If R is in the shared state, the protocol sends invalidation requests to all the sharers, reads the requested cache line from memory (since it is clean) and sends the data reply to the requester. In both cases, the protocol updates the AM bits in the directory entries of all the cache lines mapped to the requested line.

2.2 Support for Active Memory Transpose

Although we have implemented four active memory operations—*Matrix Transpose*, *Sparse Matrix Scatter/Gather*, *Linked List Linearization*, and *Memory-side Merge* [5]—for space reasons we present only matrix transpose as an example. Assume the following: A is a square matrix of dimension N, A' is a transposed matrix mapped to A, and two processors $P0$ and $P1$ access them. The cache line size is 128 bytes and the data element size is 8 bytes (one double word), so one cache line contains 16 data elements. At a certain point in time, the memory snapshot is as depicted in Figure 2 and $P0$ executes the following code:

```
for i=0 to N-1
  for j=0 to N-1
    x += A'[i][j];
```

$P0$ reads a cache line C' of A' and it misses in the data cache. C' is composed of 16 double words $w0', w1', \ldots, w15'$ that are same as $w0, w1, \ldots, w15$, and the 16 cache lines $C0, C1, \ldots, C15$ of A contain them. $P0$ and $P1$ are caching $C1$, $C2$ and $C14$ in the dirty or shared states as shown in the figure. $P0$ sends a read request to the main memory and the memory controller invokes the appropriate coherence handler.

The protocol handler reads the directory entry of C' and checks the AM bit. In this case the AM bit is set because $C1$, $C2$ and $C14$ are mapped to C' and they are cached. Therefore, instead of using the base protocol our extended protocol is invoked. From the physical address of C' the address remapping hardware (the detailed micro-architecture of the controller can be found in [5]) calculates the addresses of the cache lines mapped to C', which in this case are $C0, C1, \ldots, C15$. Since C' belongs to the re-mapped address space and the re-mapped physical address space is contiguous, from the physical address of C' the protocol can calculate the position of C' in the transposed matrix (i.e. A') if it knows the dimensions of the matrix and the size of each element of the matrix in bytes. This information along with the starting virtual address of the matrix (i.e. A) is stored in a table that is initialized at the beginning of the application via a system-level library call. Using the position of C' in A' and the starting virtual address of A the protocol calculates the virtual addresses of $C0, C1, \ldots, C15$ and looks up a TLB resident in the memory controller to compute the corresponding 16 physical addresses. Next, the protocol reads the directory entries for each of these cache lines and consults the dirty bit and the sharer vector. For $C1$, we find that it is cached by $P0$ in the dirty exclusive state. The protocol sends an intervention to $P0$ for this line because at this point $w1'$ has a stale value and the most recent value is $w1$ residing in $P0$'s cache. On receiving the reply from $P0$, the protocol updates $w1'$ with the correct value and also writes back $C1$ to memory. For $C2$, we find that it is cached by $P0$ and $P1$ in the shared state, so the protocol sends invalidations to $P0$ and $P1$ for this line. In this case, the protocol can read the correct value of $w2'$ directly from main memory. The case for $C14$ is similar to $C1$ except that $P1$, instead of $P0$, is caching this line. For the other cache lines that are clean in main memory, the protocol need not do anything. Now that the protocol has evicted all the cached lines remapped to C' from the caches of $P0$ and $P1$ and updated the data of C' with the most recent values, it is ready to reply to $P0$ with C'. Finally, the protocol updates the AM bits of all the directory entries of the cache lines re-mapped to C'. Because C' is now cached, the AM bits of $C0, C1, \ldots, C15$ are set and that of C' is clear. This guarantees correctness for future accesses to any of these cache lines.

We have described how our Matrix Transpose protocol enforces mutual exclusion between the caching states of normal and re-mapped lines. However, this is overly strict since it is legal to cache both normal and re-mapped lines provided both are in the shared state. We find though that for Matrix Transpose, enforcing mutual exclusion achieves higher performance because all transpose applications we have examined have the following sharing pattern: a processor first reads the normal space cache lines from the portion of the data set assigned to it, eventually writes to it and then moves on to read and eventually update the re-mapped space cache lines. Therefore, accesses tend to "migrate" from one space to another. When the active memory controller sees a read request for normal or re-mapped space it knows that eventually there will be an upgrade request for the same cache line. Further, there will be no access from the other space between the read and the upgrade. So our cache coherence protocol extensions choose to invalidate all the cache lines in the other space mapped to the requested line at the time of read. This keeps the upgrade handler much simpler because it does not have to worry about invalidating the shared cache lines. However, we found that Sparse Matrix Scatter/Gather does not exhibit a migratory sharing pattern, and therefore we relax the mutual exclusion constraint in that case. This illustrates an advantage of flexible memory controllers that can adapt the coherence protocol to the needs of the application.

2.3 Multi-node Protocol Extensions

This section discusses issues related to multi-node extensions of our single-node active memory protocol. Our base multi-node protocol is an MSI invalidation-based bitvector running under release consistency with a directory entry size of 8 bytes (per 128B of cache line). To reduce the occupancy at the home node, invalidation acknowledgments are collected at the requester. To reduce the number of negative acknowledgments in the system, the home node forwards writebacks to a requester whose interventions were sent too late, and the dirty third node buffers early interventions that arrive before data replies.

Our initial findings on multi-node active memory protocol extensions suggest that special care should be taken when assigning pages to home nodes. All cache lines mapped to each other should be co-located on the same node; otherwise, a request for a local memory line may require network transactions to consult the directory states of other remote lines that are mapped to it. This would complicate the protocol handlers as well as degrade performance. Further complications can arise while gathering invalidation acknowledgments on multinode systems. The active memory protocol needs to invalidate cache lines that are mapped to the requested line and cached by one or more processors. But the requested line and the lines to be invalidated have different addresses. Therefore, invalidation acknowledgment and invalidation request messages should carry different addresses or at the time of gathering invalidation acknowledgments a mapping procedure has to be invoked so that the invalidation requests get matched to the corresponding acknowledgments. Finally, we also need to give special consideration to remote interventions. While conventional protocols may have to send at most one intervention per memory request, the active memory protocol may have to send multiple interventions whose addresses are different but mapped to the requested line. Therefore, the intervention reply handler must gather all the intervention replies before replying with the requested line.

3 Protocol Evaluation

This section discusses protocol memory overhead and protocol-related performance issues. Additional memory overhead in the active memory protocol stems from an increase in protocol handler code size, directory space overhead for re-mapped cache lines, and the space required to store mapping information in a table accessible by the embedded protocol processor. As an example of the increase in handler code size, the base protocol code size is 20KB, but adding the protocol extensions to support the active memory Matrix Transpose discussed in Section 2 yields a protocol code size of 33KB. Sparse Matrix Scatter/Gather, Linked List Linearization and Memory-side Merge have protocol code sizes of 24KB, 24KB, and 26KB respectively. The directory space overhead depends on the size of the re-mapped address space. Finally, the mapping table is only 128 bytes in size. This additional memory overhead is independent of the number of nodes in the system, except that the small mapping table must be replicated on every node of the system.

There are many performance issues for active memory protocols that do not impact conventional DSM protocols. We briefly discuss some of these here. One major potential performance bottleneck is the occupancy of the memory controller. To service a request for a particular cache line, the protocol may have to consult the directory entries of all the re-mapped cache lines (16 in the previous example) and may have to take different kinds of actions based on the directory states. Performing all these directory lookups in the software running on our programmable memory controller was too slow. Instead, our controller has a specialized pipelined hardware address calculation unit that computes 16 re-mapped cache line addresses, loads directory entries in special hardware registers and initiates any necessary data memory accesses. Our software protocol handlers tightly control this active memory data unit, striking a balance between flexibility and performance. As an example of the average controller occupancy, in a single-node system for 1, 2 and 4 processors for the normal executions controller occupancy is 7.7%, 21.3% and 43.2% of the total execution time respectively, while for the active memory executions it is 15.5%, 29.4% and 47.8%, though one must remember that the active memory execution time is smaller, and therefore occupancy percentages are naturally higher.

Another important performance issue is the behavior of the directory data cache, which is accessed only by the programmable embedded processor core on the memory controller. Since an access to one directory entry may necessitate accesses to multiple re-mapped directory entries (16 in the

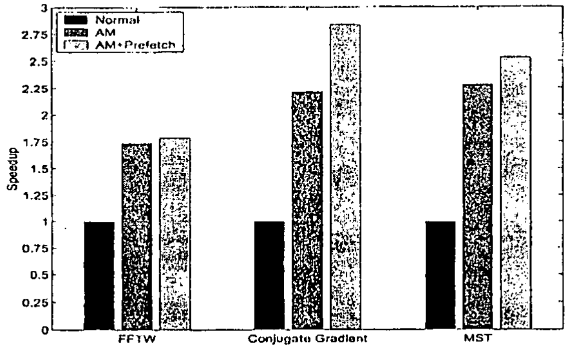

Figure 3. Active Memory Uniprocessor Speedup

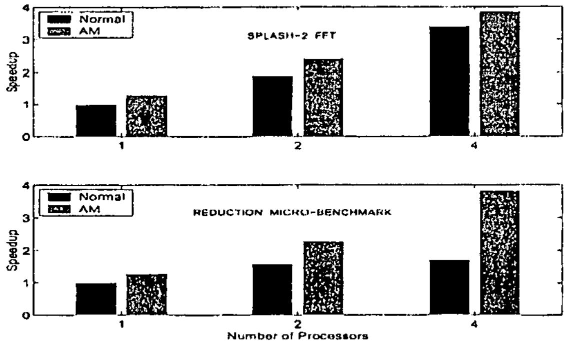

Figure 4. Active Memory Multiprocessor Speedup

above example) that may not correspond to directory entries for contiguous cache lines, the directory data cache may suffer from poor locality and hence large numbers of misses. The choice of byte-sized directory entries mitigates this problem in uniprocessor or single-node multiprocessor active memory systems. However, directory data cache performance may become an issue for extremely large problem sizes on small machines as the directory width increases for multi-node active memory systems.

Finally, an active memory protocol may have to send multiple interventions (a maximum of 16 in the above example) and every local intervention requires a data buffer that is filled by the processor-bus interface when the processor sends the intervention reply. This puts heavy pressure on the number of data buffers needed on the memory controller. We decided to keep four buffers reserved for this purpose and recycle them if a handler needs to send more than four interventions.

Table 1. L2 Cache Read Miss Count

App.	Normal	AM	Reduction Factor
FFTW	5644644	1421816	3.97
CG	13886869	3628477	3.83
MST	48582789	11829608	4.11

4 Simulation Results

We present representative simulated performance results for uniprocessor as well as single-node multiprocessor active memory systems in Figure 3 and Figure 4, respectively. FFT and FFTW use Matrix Transpose, Conjugate Gradient (CG) uses Sparse Matrix Scatter/Gather while Minimum Spanning Tree (MST) uses Linked List Linearization. The Reduction microbenchmark uses parallel reduction and shows the speedup for Memory-side Merge. Our simulator models contention in detail within the active memory controller, between the controller and its external interfaces, at main memory, and for the system bus. Further details on the simulation environment and the simulated applications can be found in [5]. In Figure 3 the "AM+Prefetch" bars correspond to the speedup achieved by our AM techniques along with exploiting new prefetching opportunities created by our AM optimizations (e.g. for Matrix Transpose it is now possible to prefetch re-mapped rows that were columns in the original matrix). Tables 1 and 2 show a comparison of L2 cache misses for Normal and non-prefetched AM executions, while Table 3 compares the data TLB miss penalty seen by the main processor for the two executions on a single processor corresponding to the results shown in Figure 3. All the tables show the reduction factor achieved by AM over normal execution for uniprocessor simulations.

For the parallel reduction microbenchmark (shown in Figure 4) the speedup of the normal application flattens out beyond two processors while the AM technique continues to achieve good speedup (3.83) for

Table 2. L2 Cache Write Miss Count

App.	Normal	AM	Reduction Factor
FFTW	5369071	1156683	4.64
CG	211323	136731	1.55
. MST	12544	8947	1.40

Table 3. Data TLB Miss Penalty in a Million Processor Cycles

App.	Normal (% of t_{exec})	AM (% of t_{exec})	Reduction Factor
FFTW	721.51 (15.72%)	13.42 (0.51%)	53.76
CG	26.63 (0.46%)	12.56 (0.48%)	2.12
MST	838.58 (4.47%)	714.03 (14.22%)	1.17

a quad-processor node. This clearly shows that even for a quad-processor node the controller occupancy does not become a bottleneck. Both uniprocessor and multiprocessor results demonstrate the clear success of our coherence-leveraged active memory technique. Further, the multiprocessor speedup shows that our active memory protocols gracefully scale as the number of processors increases.

5 Conclusions

Active memory techniques, while improving application data access patterns, introduce a data coherence problem. Since the memory controller handles every cache miss and runs the coherence protocol, it has complete control over which memory lines can be cached by a particular processor in the system and in what state. Our approach enforces the mutual exclusion between the caching states of the remapped memory lines by naturally extending the conventional DSM coherence protocol, thereby efficiently solving the coherence problem. However, the design of active n.emory protocols raises some unique issues that are quite different in nature from a conventional DSM coherence protocol. The advantage of using software coherence protocols over hardware finite state machines is that the former can support new active memory techniques without changes to the memory controller hardware. This paper presents representative results on uniprocessors and single-node multiprocessors that confirm that our approach scales and performs well. Further, this protocol extension naturally lends itself to the research and development of multinode active memory systems that we call Active Memory Clusters [3], which have the ability to attain hardware DSM performance on commodity clusters.

Acknowledgments

This research was supported by Cornell's Intelligent Information Systems Institute and NSF CAREER Award CCR-9984314.

References

[1] J. B. Carter et al. Impulse: Building a Smarter Memory Controller. In *Proceedings of the Fifth International Symposium on High Performance Computer Architecture*, January 1999.

[2] M. Hall et al. Mapping Irregular Applications to DIVA, A PIM-based Data-Intensive Architecture. *Supercomputing*, Portland, OR, Nov. 1999.

[3] M. Heinrich, E. Speight, and M. Chaudhuri. Active Memory Clusters: Efficient Multiprocessing on Commodity Clusters. In *Proceedings of the Fourth International Symposium on High-Performance Computing, Lecture Notes in Computer Science*, Springer-Verlag, May 2002.

[4] Y. Kang et al. FlexRAM: Toward an Advanced Intelligent Memory System. *International Conference on Computer Design*, October 1999.

[5] D. Kim, M. Chaudhuri, and M. Heinrich. Leveraging Cache Coherence in Active Memory Systems. In *Proceedings of the 16th ACM International Conference on Supercomputing*, New York City, June 2002.

[6] J. Kuskin et al. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 302–313, April 1994.

[7] M. Oskin, F. T. Chong, and T. Sherwood. Active Pages: A Computation Model for Intelligent Memory. In *Proceedings of the 25th International Symposium on Computer Architecture*, 1998.